Lecture 1 R Basics
Section A & B

Ziwen Ye

Stevens Institute of Technology

zye2@stevens.edu

August 26, 2018

Agenda

1. Data Types
   - Atomic Classes
   - Vector
   - Matrix
   - List
2. Control Structures
   - for
   - if-else
   - while
3. Example

Atomic Classes

R has five basic or "atomic" classes of objects:

- numeric (real number)
- integer
- character (or string)
- complex
- logical (T/F)

A vector only support one type of object.
For example, 3.1, 4.2 or 'a', 'b', are actually **one-element vectors**.
We use some embedded R functions, such as *mode()*, *typeof()*, *storage.mode()* to inspect which mode the variable belongs to.
When combining two

Atomic Classes

The results of modes and storage modes for the different vector types are listed in the following table.

typeof	**mode**	**storage.mode**
logical	logical	logical
integer	numeric	integer
double	numeric	double
complex	complex	complex
character	character	character

Table: Vector Types

Explicit Coercion

R objects are often coerced to different types during computations. We do explicit coercion using **as.*** functions, if available.

Example (Explicit Coercion)

```
> x <- -3:3
> x
[1] -3 -2 -1  0  1  2  3
> typeof(x)
[1] "integer"
> as.numeric(x)
[1] -3 -2 -1  0  1  2  3
> as.logical(x)
[1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
> as.character(x)
[1] "-3" "-2" "-1" "0"  "1"  "2"  "3"
```

Vector

Vector: the workhorse in R

- The simplest data structure in R.
- All elements of a vector must have the same mode.

Example (Creating Vectors)

```
> x <- c(4, 5, 6)          # numeric
> x <- c(TRUE, FALSE)      # logical
> x <- c(T, F)             # logical
> x <- c('a', 'b', 'c')    # character
> x <- 1:100               # numeric
> x <- c(1+0i, 3+5i)       # complex
```

Vector

There are three simple ways to create vectors:

- c()
- seq()
- rep()

Vector

Example (Creating Vectors)

```
> x <- seq(10, 20, by = 2)
> x
[1] 10 12 14 16 18 20
> y <- rep(x = c(1, 2, 3), 2)
> y
[1] 1 2 3 1 2 3
> z <- c(x, y)
> z
[1] 10 12 14 16 18 20  1  2  3  1  2  3
```

Matrix

Technically, a matrix is just a vector with two subscripts: the number of rows and the number of columns.

Example

```
> x <- 1:9
> matrix(x, nrow = 3, ncol = 3)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> matrix(x, nrow = 3, ncol = 3, byrow=T)     # byrow
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

Matrix

Example (Matrix cont'd)

```
> x <- 1:3
> y <- 4:6
> m1 <- rbind(x, y)
> m1
  [,1] [,2] [,3]
x    1    2    3
y    4    5    6
> m2<- cbind(x, y)
> m2
     x y
[1,] 1 4
[2,] 2 5
[3,] 3 6
```

Matrix

Example (Transpose)

```
> m <- matrix(1:9, nrow = 3, byrow = T)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> t(m)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Matrix

Example (Matrix Inverse)

```
# the inverse of m
> solve(m)    # you can't, m is singular
Error in solve.default(m) :
  system is computationally singular:
  reciprocal condition number = 2.59052e-18
> m[1, 1] = 10
> m[2, 2] = 8
> solve(m)
            [,1]       [,2]        [,3]
[1,]  0.13333333  0.03333333 -0.06666667
[2,]  0.03333333  0.38333333 -0.26666667
[3,] -0.13333333 -0.36666667  0.40000000
```

List

- Lists are special type of vector that can contain elements of different types.
- Similar to a *dictionary* in Python, or a *struct* in C.
- Very important, forming the basis for data frames, object-oriented programming. (later)

List

Example (Createing Lists)

```
> l <- list("John", 12345, "Male")
> l
[[1]]
[1] "John"

[[2]]
[1] 12345

[[3]]
[1] "Male"
```

List

Lists can have names, and you access members by '$'.

Example (List with names)

```
> l <- list(name = "John", ID = 12345, gender = "Male")
> l
$name
[1] "John"
$ID
[1] 12345
$gender
[1] "Male"
> l$name    # access list member by '$'
[1] "John"
> l$ID
[1] 12345
```

Subsetting

Example (Subsetting)

```
> x <- 1:10
> m <- matrix(x, nrow = 2, byrow = T)

> y <- c('a', 'b', 'c')
> l <- list(numbers = x, chars = y)

> x[3]              # subsetting of a vector
[1] 3
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
> m[2, 2]           # subsetting of a matrix
[1] 7
```

Subsetting

Example (Subsetting)

```
> l
$numbers
 [1]  1  2  3  4  5  6  7  8  9 10
$chars
[1] "a" "b" "c"
> l[[1]]              # subsetting of a list
 [1]  1  2  3  4  5  6  7  8  9 10
> l[["numbers"]]
 [1]  1  2  3  4  5  6  7  8  9 10
> l[[1]][3]           # nested subsetting
[1] 3
> l[[c(1, 3)]]
[1] 3
```

Summary

Example

```
> # create variables
> x <- 1:10                                  # vector
> y <- c('a', 'b', 'c')
> m <- matrix(x, nrow = 2, byrow = T)        # matrix
> l <- list(numbers = x, chars = y)          # list

> # subsetting
> x[3]                  # subsetting of a vector
> m[2, 2]               # subsetting of a matrix
> l[[1]]                # subsetting of a list
> l[["numbers"]]
> l[[1]][3]             # nested subsetting
> l[[c(1, 3)]]
```

for loop

For loops take an iterator variable and assign it successive values from a vector. For loops are most commonly used for iterating over the elements of an object (vector, list, etc).

Example

```
x <- 1:5
for(i in x)
{
    print(i)    # same thing
}

for (i in 1:5)
{
    print(i)    # every iteration increases 1
}
```

for loop

Example (for loop cont'd)

```
# calculate 1 + 2 + 3 + ... + 100 = 5050
> result <- 0
> for (i in 1:100)
+ {
+     result = result + i
+ }
> result
[1] 5050
```

for loop

Example (Nested loops)

```
> m2 <- matrix(NA, nrow = 3, ncol = 4)          # NA matrix
> m1 <- matrix(1:12, nrow = 3, byrow = T)
> for (i in 1:3)       # i is row index
+ {
+     for (j in 1:4)  # j is col index
+     {
+         m2[i, j] <- m1[i, j] * 10
+     }
+ }
> m2
     [,1] [,2] [,3] [,4]
[1,]   10   20   30   40
[2,]   50   60   70   80
[3,]   90  100  110  120
```

if-else statement

Conditional statements are features of a programming language which perform different computations or actions depending on whether a programmer-specified boolean condition evaluates to true or false.

Example (if-else)

```
> x <- F
> if (x) # x needs to be a logical value
+ {
+     "x is TRUE"
+ } else {
+     "x is FALSE"
+ }
[1] "x is FALSE"
# or ...
> ifelse(x, "x is TRUE", "x is FALSE")
[1] "x is FALSE"
```

if-else statement

Another if-else example.

Example (if-else cont'd)

```
x <- 1:10
if (length(x) > 15)
{
    print("x is a long array")
} else {
    print("x is short array")
}
```

while loop

Another way of doing iteration: while loop.

Example (while loop)

```
while(...)  # condition
{
    # statement
}
```

while loop

Example

```
result <- 0
while(TRUE)     # infinite loop
{
    print(result)
    result = result + 1
}

result <- 0
while(result < 1000
{
    print(result)
    result = result + 1
}
```

while loop

Example

```
result <- 0
sum <- 0
while (result <= 100)
{
    if (sum > 3000)      # break loop using if statement
        break
    sum = sum + result
    print(sum)
    result = result + 1
}
```

Exercise: Fibonacci

You may heard the Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13...
where $S_n = S_{n-1} + S_{n-2}$, now implement this in R